

03 | 序列化：对象怎么在网络中传输？

2020-02-24 何小锋

RPC实战与核心原理

[进入课程 >](#)



讲述：张浩

时长 14:18 大小 13.10M



你好，我是何小锋。上一讲我讲解了在 RPC 框架中，如何设计可扩展的、向后兼容的协议，其关键点就是利用好 Header 中的扩展字段以及 Payload 中的扩展字段，通过扩展字段向后兼容。

那么承接上一讲的一个重点，今天我会讲解下 RPC 框架中的序列化。要知道，在不同的场景下合理地选择序列化方式，对提升 RPC 框架整体的稳定性和性能是至关重要的。

为什么需要序列化？

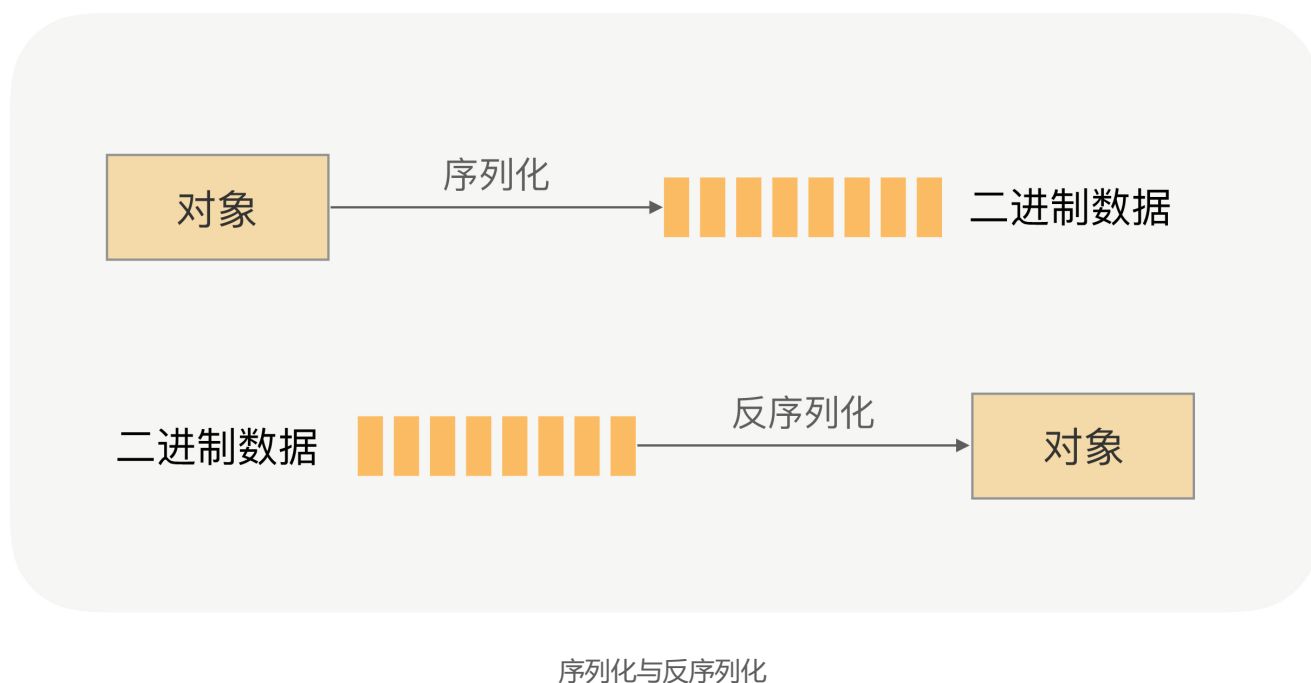


首先，我们得知道什么是序列化与反序列化。

我们先回顾下 [\[第 01 讲\]](#) 介绍过的 RPC 原理的内容，在描述 RPC 通信流程的时候我说过：

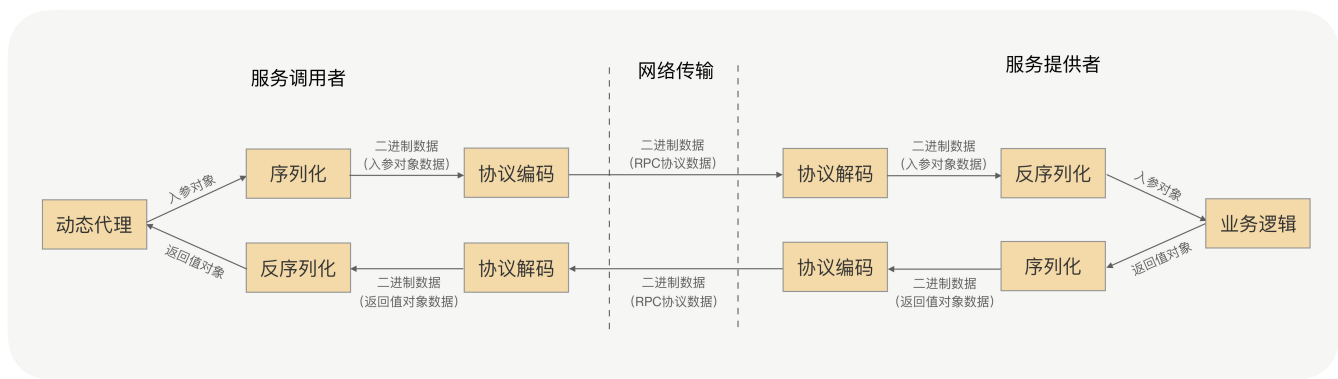
网络传输的数据必须是二进制数据，但调用方请求的出入参数都是对象。对象是不能直接在网络中传输的，所以我们需要提前把它转成可传输的二进制，并且要求转换算法是可逆的，这个过程我们一般叫做“序列化”。这时，服务提供方就可以正确地从二进制数据中分割出不同的请求，同时根据请求类型和序列化类型，把二进制的消息体逆向还原成请求对象，这个过程我们称之为“反序列化”。

这两个过程如下图所示：



总结来说，序列化就是将对象转换成二进制数据的过程，而反序列就是反过来将二进制转换为对象的过程。

那么 RPC 框架为什么需要序列化呢？还是请你回想下 RPC 的通信流程：



RPC通信流程图

不妨借用个例子帮助你理解，比如发快递，我们要发一个需要自行组装的物件。发件人发之前，会把物件拆开装箱，这就好比序列化；这时候快递员来了，不能磕碰呀，那就要打包，这就好比将序列化后的数据进行编码，封装成一个固定格式的协议；过了两天，收件人收到包裹了，就会拆箱将物件拼接好，这就好比是协议解码和反序列化。

所以现在你清楚了吗？因为网络传输的数据必须是二进制数据，所以在RPC调用中，对入参对象与返回值对象进行序列化与反序列化是一个必须的过程。

有哪些常用的序列化？

那这么看来，你会不会觉得这个过程很简单呢？实则不然，很复杂。我们可以先看看都有哪些常用的序列化，下面我来简单地介绍下几种常用的序列化方式。

JDK 原生序列化

如果你会使用Java语言开发，那么你一定知道JDK原生的序列化，下面是JDK序列化的一个例子：

复制代码

```
1 import java.io.*;
2
3 public class Student implements Serializable {
4     //学号
5     private int no;
6     //姓名
7     private String name;
8
9     public int getNo() {
10         return no;
11     }
12 }
```

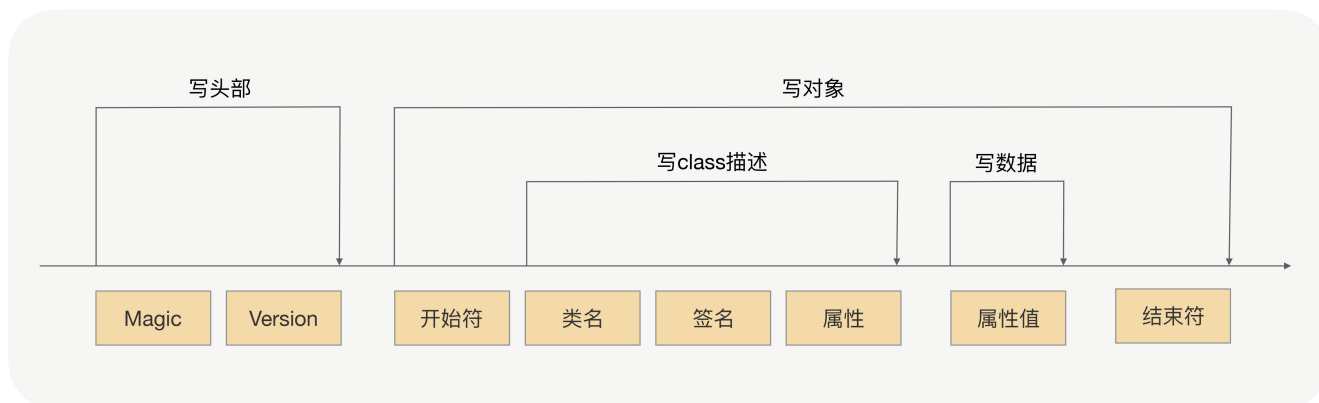
```

13     public void setNo(int no) {
14         this.no = no;
15     }
16
17     public String getName() {
18         return name;
19     }
20
21     public void setName(String name) {
22         this.name = name;
23     }
24
25     @Override
26     public String toString() {
27         return "Student{" +
28             "no=" + no +
29             ", name='" + name + '\'' +
30             '}';
31     }
32
33     public static void main(String[] args) throws IOException, ClassNotFoundException {
34         String home = System.getProperty("user.home");
35         String basePath = home + "/Desktop";
36         FileOutputStream fos = new FileOutputStream(basePath + "student.dat");
37         Student student = new Student();
38         student.setNo(100);
39         student.setName("TEST_STUDENT");
40         ObjectOutputStream oos = new ObjectOutputStream(fos);
41         oos.writeObject(student);
42         oos.flush();
43         oos.close();
44
45         FileInputStream fis = new FileInputStream(basePath + "student.dat");
46         ObjectInputStream ois = new ObjectInputStream(fis);
47         Student deStudent = (Student) ois.readObject();
48         ois.close();
49
50         System.out.println(deStudent);
51
52     }
53 }

```

我们可以看到，JDK 自带的序列化机制对使用者而言是非常简单的。序列化具体的实现是由 `ObjectOutputStream` 完成的，而反序列化的具体实现是由 `ObjectInputStream` 完成的。

那么 JDK 的序列化过程是怎样完成的呢？我们看下下面这张图：



ObjectOutputStream序列化过程图

序列化过程就是在读取对象数据的时候，不断加入一些特殊分隔符，这些特殊分隔符用于在反序列化过程中截断用。

头部数据用来声明序列化协议、序列化版本，用于高低版本向后兼容

对象数据主要包括类名、签名、属性名、属性类型及属性值，当然还有开头结尾等数据，除了属性值属于真正的对象值，其他都是为了反序列化用的元数据

存在对象引用、继承的情况下，就是递归遍历“写对象”逻辑

实际上任何一种序列化框架，核心思想就是设计一种序列化协议，将对象的类型、属性类型、属性值——按照固定的格式写到二进制字节流中来完成序列化，再按照固定的格式——读出对象的类型、属性类型、属性值，通过这些信息重新创建出一个新的对象，来完成反序列化。

JSON

JSON 可能是我们最熟悉的一种序列化格式了，JSON 是典型的 Key-Value 方式，没有数据类型，是一种文本型序列化框架，JSON 的具体格式和特性，网上相关的资料非常多，这里就不再介绍了。

他在应用上还是很广泛的，无论是前台 Web 用 Ajax 调用、用磁盘存储文本类型的数据，还是基于 HTTP 协议的 RPC 框架通信，都会选择 JSON 格式。

但用 JSON 进行序列化有这样两个问题，你需要格外注意：

JSON 进行序列化的额外空间开销比较大，对于大数据量服务这意味着需要巨大的内存和磁盘开销；

JSON 没有类型，但像 Java 这种强类型语言，需要通过反射统一解决，所以性能不会太好。

所以如果 RPC 框架选用 JSON 序列化，服务提供者与服务调用者之间传输的数据量要相对较小，否则将严重影响性能。

Hessian

Hessian 是动态类型、二进制、紧凑的，并且可跨语言移植的一种序列化框架。Hessian 协议要比 JDK、JSON 更加紧凑，性能上要比 JDK、JSON 序列化高效很多，而且生成的字节数也更小。

使用代码示例如下：

 复制代码

```
1 Student student = new Student();
2 student.setNo(101);
3 student.setName("HESSIAN");
4
5 //把student对象转化为byte数组
6 ByteArrayOutputStream bos = new ByteArrayOutputStream();
7 Hessian2Output output = new Hessian2Output(bos);
8 output.writeObject(student);
9 output.flushBuffer();
10 byte[] data = bos.toByteArray();
11 bos.close();
12
13 //把刚才序列化出来的byte数组转化为student对象
14 ByteArrayInputStream bis = new ByteArrayInputStream(data);
15 Hessian2Input input = new Hessian2Input(bis);
16 Student deStudent = (Student) input.readObject();
17 input.close();
18
19 System.out.println(deStudent);
```

相对于 JDK、JSON，由于 Hessian 更加高效，生成的字节数更小，有非常好的兼容性和稳定性，所以 Hessian 更加适合作为 RPC 框架远程通信的序列化协议。

但 Hessian 本身也有问题，官方版本对 Java 里面一些常见对象的类型不支持，比如：

Linked 系列，LinkedHashMap、LinkedHashSet 等，但是可以通过扩展 CollectionDeserializer 类修复；

Locale 类，可以通过扩展 ContextSerializerFactory 类修复；

Byte/Short 反序列化的时候变成 Integer。

以上这些情况，你在实践时需要格外注意。

Protobuf

Protobuf 是 Google 公司内部的混合语言数据标准，是一种轻便、高效的结构化数据存储格式，可以用于结构化数据序列化，支持 Java、Python、C++、Go 等语言。Protobuf 使用的时候需要定义 IDL（Interface description language），然后使用不同语言的 IDL 编译器，生成序列化工具类，它的优点是：

序列化后体积相比 JSON、Hessian 小很多；

IDL 能清晰地描述语义，所以足以帮助并保证应用程序之间的类型不会丢失，无需类似 XML 解析器；

序列化反序列化速度很快，不需要通过反射获取类型；

消息格式升级和兼容性不错，可以做到向后兼容。

使用代码示例如下：

```
1  /**
2   *
3   * // IDL 文件格式
4   * syntax = "proto3";
5   * option java_package = "com.test";
6   * option java_outer_classname = "StudentProtobuf";
7   *
8   * message StudentMsg {
9   * //序号
10  * int32 no = 1;
11  * //姓名
12  * string name = 2;
13
```

 复制代码

```
14  * }
15  *
16  */
17
18  StudentProtobuf.StudentMsg.Builder builder = StudentProtobuf.StudentMsg.newBuilder()
19  builder.setNo(103);
20  builder.setName("protobuf");
21
22  //把student对象转化为byte数组
23  StudentProtobuf.StudentMsg msg = builder.build();
24  byte[] data = msg.toByteArray();
25
26  //把刚才序列化出来的byte数组转化为student对象
27  StudentProtobuf.StudentMsg deStudent = StudentProtobuf.StudentMsg.parseFrom(data);
28
29  System.out.println(deStudent);
```

Protobuf 非常高效，但是对于具有反射和动态能力的语言来说，这样用起来很费劲，这一点就不如 Hessian，比如用 Java 的话，这个预编译过程不是必须的，可以考虑使用 Protostuff。

Protostuff 不需要依赖 IDL 文件，可以直接对 Java 领域对象进行反 / 序列化操作，在效率上跟 Protobuf 差不多，生成的二进制格式和 Protobuf 是完全相同的，可以说是一个 Java 版本的 Protobuf 序列化框架。但在使用过程中，我遇到过一些不支持的情况，也同步给你：

不支持 null；

ProtoStuff 不支持单纯的 Map、List 集合对象，需要包在对象里面。

RPC 框架中如何选择序列化？

我刚刚简单地介绍了几种最常见的序列化协议，其实远不止这几种，还有 Message pack、kryo 等。那么面对这么多的序列化协议，在 RPC 框架中我们该如何选择呢？

首先你可能想到的是性能和效率，不错，这的确是一个非常值得参考的因素。我刚才讲过，序列化与反序列化过程是 RPC 调用的一个必须过程，那么序列化与反序列化的性能和效率势必将直接关系到 RPC 框架整体的性能和效率。

那除了这点，你还想到了什么？

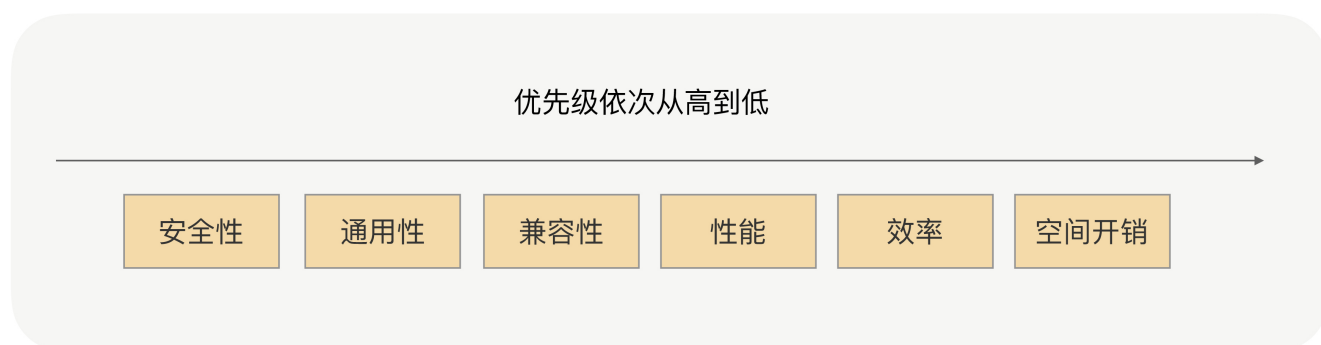
对，还有空间开销，也就是序列化之后的二进制数据的体积大小。序列化后的字节数据体积越小，网络传输的数据量就越小，传输数据的速度也就越快，由于 RPC 是远程调用，那么网络传输的速度将直接关系到请求响应的耗时。

现在请你再想想，还有什么因素可以影响到我们的选择？

没错，就是序列化协议的通用性和兼容性。在 RPC 的运营中，序列化问题恐怕是我碰到的和解答过的最多的问题了，经常有业务会向我反馈这个问题，比如某个类型为集合类的入参服务调用者不能解析了，服务提供方将入参类加一个属性之后服务调用方不能正常调用，升级了 RPC 版本后发起调用时报序列化异常了...

在序列化的选择上，与序列化协议的效率、性能、序列化协议后的体积相比，其通用性和兼容性的优先级会更高，因为他是会直接关系到服务调用的稳定性和可用率的，对于服务的性能来说，服务的可靠性显然更加重要。我们更加看重这种序列化协议在版本升级后的兼容性是否很好，是否支持更多的对象类型，是否是跨平台、跨语言的，是否有很多人已经用过并且踩过了很多的坑，其次我们才会去考虑性能、效率和空间开销。

还有一点我要特别强调。除了序列化协议的通用性和兼容性，序列化协议的安全性也是非常重要的一个参考因素，甚至应该放在第一位去考虑。以 JDK 原生序列化为例，它就存在漏洞。如果序列化存在安全漏洞，那么线上的服务就很可能被入侵。



综合上面几个参考因素，现在我们来总结一下这几个序列化协议。

我们首选的还是 Hessian 与 Protobuf，因为他们在性能、时间开销、空间开销、通用性、兼容性和安全性上，都满足了我们的要求。其中 Hessian 在使用上更加方便，在对象的兼容性上更好；Protobuf 则更加高效，通用性上更有优势。

RPC 框架在使用时要注意哪些问题？

了解了在 RPC 框架中如何选择序列化，那么我们在使用过程中需要注意哪些序列化上的问题呢？

我刚才讲过，在 RPC 的运营中，我遇到的最多的问题就是序列化问题了，除了早期 RPC 框架本身出现的问题以外，大多数问题都是使用方使用不正确导致的，接下来我们就盘点下这些高频出现的人为问题。

对象构造得过于复杂：属性很多，并且存在多层的嵌套，比如 A 对象关联 B 对象，B 对象又聚合 C 对象，C 对象又关联聚合很多其他对象，对象依赖关系过于复杂。序列化框架在序列化与反序列化对象时，对象越复杂就越浪费性能，消耗 CPU，这会严重影响 RPC 框架整体的性能；另外，对象越复杂，在序列化与反序列化的过程中，出现问题的概率就越高。

对象过于庞大：我经常遇到业务过来咨询，为啥他们的 RPC 请求经常超时，排查后发现他们的入参对象非常得大，比如为一个大 List 或者大 Map，序列化之后字节长度达到了上兆字节。这种情况同样会严重地浪费了性能、CPU，并且序列化一个如此大的对象是很耗费时间的，这肯定会直接影响到请求的耗时。

使用序列化框架不支持的类作为入参类：比如 Hessian 框架，他天然是不支持 LinkHashMap、LinkedHashSet 等，而且大多数情况下最好不要使用第三方集合类，如 Guava 中的集合类，很多开源的序列化框架都是优先支持编程语言原生的对象。因此如果入参是集合类，应尽量选用原生的、最为常用的集合类，如 HashMap、ArrayList。

对象有复杂的继承关系：大多数序列化框架在序列化对象时都会将对象的属性——进行序列化，当有继承关系时，会不停地寻找父类，遍历属性。就像问题 1 一样，对象关系越复杂，就越浪费性能，同时又很容易出现序列化上的问题。

在 RPC 框架的使用过程中，我们要尽量构建简单的对象作为入参和返回值对象，避免上述问题。

总结

今天我们深入学习了什么是序列化，并介绍了如 JDK 原生序列化、JSON、Hessian 以及 Protobuf 等几种常见的序列化方式。

除了这些基础知识之外，我们重点讲解了在 RPC 框架中如何去选择序列化协议，我们有这样几个很重要的参考因素，优先级从高到低依次是安全性、通用性和兼容性，之后我们会再考虑序列化框架的性能、效率和空间开销。

这归根结底还是因为服务调用的稳定性与可靠性，要比服务的性能与响应耗时更加重要。另外对于 RPC 调用来说，整体调用上，最为耗时、最消耗性能的操作大多都是服务提供者执行业务逻辑的操作，这时序列化的开销对于服务整体的开销来说影响相对较小。

在使用 RPC 框架的过程中，我们构造入参、返回值对象，主要记住以下几点：

1. 对象要尽量简单，没有太多的依赖关系，属性不要太多，尽量高内聚；
2. 入参对象与返回值对象体积不要太大，更不要传太大的集合；
3. 尽量使用简单的、常用的、开发语言原生的对象，尤其是集合类；
4. 对象不要有复杂的继承关系，最好不要有父子类的情况。

实际上，虽然 RPC 框架可以让我们发起全程调用就像调用本地，但在 RPC 框架的传输过程中，入参与返回值的根本作用就是用来传递信息的，为了提高 RPC 调用整体的性能和稳定性，我们的入参与返回值对象要构造得尽量简单，这很重要。

课后思考

RPC 框架在序列化框架的选型上，你认为还需要考虑哪些因素？你还知道哪些优秀的序列化框架，它们又是否适合在 RPC 调用中使用？

欢迎留言和我分享你的答案和经验，也欢迎你把文章分享给你的朋友，邀请他加入学习。我们下节课再见！

更多课程推荐

RPC 实战与核心原理

高效解决分布式系统的通信难题

何小锋

京东技术架构部首席架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 02 | 协议：怎么设计可扩展且向后兼容的协议？

精选留言 (15)

 写留言



盘胧

2020-02-24

请问如果对象类继承关系比较简单，使用组合后，这个类里组合的其他类对象实例也需要连续被序列化么？

作者回复：会嵌套序列化



 2



Sephiroth

2020-02-24

为什么没有说thrift呢？

展开 ∨

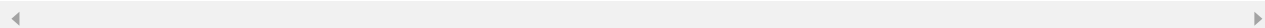


艾斯曼

2020-02-24

还有thrift框架，它和grpc差不多。最早的rpc还有基于xml的，这个就更差了，不过早期确实好有用。

作者回复: 是的，rpc也在不断发展



小豆角

2020-02-25

如果要传输的数据只有一个字节，也不需要序列化了。凡是离开内存需要传输或者持久化的数据，都要先进行序列化。数据离开内存，进行io的时候，就应该学习序列化的概念了。序列化就是流化，数据转成字节流。这样认为对吗

展开 ∨



旺财勇士

2020-02-25

课程非常棒，就是更新有点慢

展开 ∨



忆水寒

2020-02-25

既要考虑序列化/反序列化的高效性 也要考虑压缩的性能，也要考虑平台的兼容性。当然了，将元素进行二进制编码也是可以传输的。

展开 ∨



阿卧

2020-02-25

1. 在不是太在意性能的场景下，考虑其可读性。目前json序列化用的比较多。
老师，为什么hessian不支持linkedhashmap这样的对象啊？



陈国林

2020-02-24

老师你好，请教一个问题。【如果RPC框架基于HTTP协议，并且使用JSON做为序列化协议。那么业务请求数据在网络传输的过程中是JSON格式还是二进制格式呢？或者说 OSI 7 层网络模型是否会自动进行一些数据的序列化？】 谢谢~

展开 ∨



Jackey

2020-02-24

想到了Redis使用的RESP，在做序列化时也是会增加很多冗余的字符，但它胜在实现简单、可读性强易于理解



小伟

2020-02-24

何老师，为何数据在网络传输时一定要是二进制？
我想主要还是兼容、高效的考虑吧。



小伟

2020-02-24

IDL文件我们最早大概是在2000年左右，用来解耦Java和Corba的，它的问题是每次改动后的版本控制比较麻烦，因为Java端和Corba端要拿相同版本的IDL才能编过。
Protobuf里是如何让收发两端都拿到同一版本的IDL呢？

展开 ∨



楼下小黑哥

2020-02-24

我觉得可读性是否也该考虑。

总结下序列化协议可以分为两类

1.文本类序列化方式，如 xml，json。优点就是可读性好，构造方便，调试也简单。不过缺点也明显，传输体积大，性能差。

2.二进制类序列化方式，如 Hessian，Protobuf，优点性能好。...

展开 ∨



鱼_XueTr

2020-02-24

flatbuffer算么？

展开 ▾



小兜

2020-02-24

在序列化框架的选择上，由于用于数据传输所以还需要考虑稳定性。Thrift也会用于rpc的序列化。

作者回复: 根据实际情况选择



小美

2020-02-24

Protostuff对象属性，以及容器内元素都支持null吧，请问老师说的不支持是指在什么场景？

作者回复: 前几天刚看到最新发布的版本已经改进支持了，原来序列化对象数组空元素有问题。

